Termination of Ethereum's Smart Contracts *

Thomas Genet[1], Thomas Jensen[1], and Justine Sauvage[1]

[1] *Univ Rennes, Inria, CNRS, IRISA.*

Abstract

Ethereum is a decentralized blockchain technology equipped with so-called Smart Contracts. A contract is a program whose code is public, which can be triggered by any user, and whose actual execution is performed by miners participating in Ethereum. Miners execute the contract on the Ethereum Virtual Machine (EVM) and apply its effect by adding new blocks to the blockchain. A contract that takes too much time to be processed by the miners of the network may result into delays or a denial of service in the Ethereum system. To prevent this scenario, termination of Ethereum's Smart Contracts is ensured using a gas mechanism. Roughly, the EVM consumes gas to process each instruction of a contract and the gas provided to run a contract is limited. This technique could make termination of contracts easy to prove but the way the official definition of the EVM specifies gas usage makes the proof of this property non-trivial. EVM implementations and formal analysis techniques of EVM's Smart Contracts use termination of contracts as an assumption, so having a formal proof of termination of contracts is crucial. This paper presents a mechanized, formal, and general proof of termination of Smart Contracts based on a measure of EVM call stacks.

A blockchain is a decentralized ledger, shared over a network, on which all users agree. Users can submit new elements to be added to this ledger. To add new elements in the ledger, one needs to add a new block (containing the new elements) to the blockchain. A block will be added to the blockchain if most of the participants agree on it. In Bitcoin, to add a new block to the chain, one has to solve a cryptographic puzzle *on this new block* in a limited amount of time (around 10 minutes in Bitcoin). This is called *mining a block*. Since the puzzle is computationally difficult it requires that most users participate in its resolution. Users contributing to the resolution are called miners. The fact that most miners try to solve the same puzzle entails that they *all agree* on the block itself and on all the added elements.

Bitcoin is equipped with a programming language, called Script (script, 2014), that is used to define programs reading inputs in the blockchain and proposing outputs (new elements) to be added to the blockchain. It is the role of the miners to execute the Script programs and to build the new blocks containing the outputs of those programs. If *one* Script program is non-terminating, this prevents miners from building new blocks and adding them to the blockchain within the 10 minutes time limit. If *many* Script programs are non terminating,

*This work was partially supported by Laboratoire d'excellence CominLabs

this could cause a denial of service in the Bitcoin system. This is the reason why the Script language is not Turing-complete, in particular it has no loops.

Ethereum extends Bitcoin's blockchain with a Turing-complete programming language and the ability to store those programs (called contracts) in the blockchain itself. Contracts are programmed into dedicated high-level languages like Solidity (solidity, 2014) or Vyper (vyper, 2017) and compiled to a bytecode format executed by the so-called Ethereum Virtual Machine (EVM). Since the programming language is Turing-complete, Ethereum needs to prevent looping contracts. In addition, Ethereum also targets to accelerate the pace of block additions w.r.t. Bitcoin. Thus, a terminating contract that takes too long to complete is another source of denial of service for Ethereum. Ethereum protects its system from non terminating programs and too complex programs with a single mechanism: the gas (Buterin, 2013). Intuitively, the EVM consumes gas to process each instruction of a contract and the gas provided to run a contract is limited.

Though this mechanism looks simple and robust, the protection it offers against denial of service is fragile. For instance, in 2016, badly chosen gas values for some EVM instructions resulted into several denial of service of Ethereum. This had to be fixed by two consecutive hard forks of the system (Hudson, 2016a; Hudson, 2016b). Independently of choosing for the best gas cost for each instruction, a *general* question to ask is whether the gas mechanism is sufficient to prove termination of *any* contract? Surprisingly, *proving formally* that this is true is not trivial because of the complexity of the EVM semantics (see Section 4).

The goal of this paper is twofold: to prove that no program can execute indefinitely without consuming gas in the EVM execution model, and to prove it in a way that can be used in a mechanized proof. More precisely, we present two termination proofs on two slightly different EVM semantics. The first model is the formal semantics of the (foundational) Ethereum Yellow Paper (Gavin, 2014), the Isabelle/HOL EVM semantics (Hirai, 2017; Amani et al., 2018) and the small-step formal semantics of (Grishchenko et al., 2018b). The second model is the semantics of the reference implementations of EVM such as (pevm, 2017; gevm, 2014). Noteworthy, the implementations and the Yellow Paper disagree on the gas consumption when calling a contract from another contract. In the Yellow Paper, when a contract c_1 calls another contract c_2 with, say, g units of gas, the gas associated to c_1 *is not charged immediately*. In implementations, this gas is immediately consumed. This little difference in the semantics makes a big difference when we are interested in proving the termination of contracts. Indeed, with the Yellow Paper semantics, a contract c_1 calling itself can loop without consuming gas, until it exhausts the call stack. This paper provides a termination proof of contracts for the two semantics. Proving termination of contracts when gas is charged immediately is natural and will be briefly discussed in Section 6. Proving termination of the contracts for the Yellow Paper semantics is more difficult and requires a complex termination measure on call stacks. Though the Yellow Paper semantics differs from the reference implementations, having a termination proof for this semantics is important. First, this termination proof contributes evidence that the Yellow Paper semantic model is indeed coherent. Second, this semantics serves as a base for formal verification tools, such as (Grishchenko et al., 2018b; Grishchenko et al., 2018a), or for formal semantics such as (Hirai, 2017; Amani et al., 2018). In those tools and

semantics, the termination of contracts is used as an assumption. In particular, in the Isabelle/HOL formalisation of (Hirai, 2017; Amani et al., 2018) the termination of the contract evaluation is proven using an internal step counter, which is not related to the gas, and simplifies the proofs.[1] Our proof complement their work by showing how the gas itself ensures termination of contracts, and thus assuming termination of contracts in the Yellow Paper semantics was indeed correct.

Contributions: This paper gives the first formal and mechanized proof of termination of EVM contracts, written in EVM bytecode. The central part is a measure that can be used for the proof of termination in a proof assistant (in our case Isabelle/HOL). We prove termination for:

- the two variants of the semantics of the contract call described above;
- a formal model where contracts can add and run arbitrary new contracts;
- a formal model that safely over-approximates the EVM semantics with minimal assumptions. In particular, for non-zero cost byte code operations (i.e. all operations except STOP, RETURN, REVERT), we only require that they have *any* strictly positive cost. Similarly, we only require the call stack size is upper-bounded by *any* natural number greater than 0.

Note that having minimal assumptions on the concrete gas costs for each operation is valuable because the gas cost has already changed several times during the EVM's lifetime[2] and is likely to evolve again since gas pricing of operations is still not fully satisfactory (Yang et al., 2019).

1 Related work

The Ethereum system has been formalized in the so-called Yellow Paper (Gavin, 2014) which has been updated recently (Gavin, 2019). This update does not impact gas consumption but provides some new instructions which are taken into account in our formal proof. A nice complementary reading is the White Paper (Buterin, 2013) which provides useful intuitions about the system. There are several available reference implementations of EVM such as (pevm, 2017; gevm, 2014). As explained above, implementations and the Yellow Paper disagree on the gas consumption of the call operations.

Grishchenko et al. have proposed EtherTrust (ethertrust, 2017) a verification framework for the static analysis of contracts code (Grishchenko et al., 2018a). The static analysis tools focus on proving some security properties on contracts, such as single-entrancy (Grishchenko et al., 2018b). EtherTrust comes with a complete small-step semantics for EVM (Grishchenko et al., 2018b) that uses the Yellow Paper semantics for the contract call.

[1] In the comments of the `lem/evm.lem` specification file, it becomes evident that the termination proof uses an artificial step counter and not the gas mechanism. This choice was made to simplify the proof as stated line 1859 of lem/evm.lem (FEL, 2018).

[2] There was a cost increase for 8 EVM instructions on 2016/10/18 (Hudson, 2016a) and a cost increase for one EVM instruction on 2016/11/18 (Hudson, 2016b)

There are several attempts to define a mechanized and formal semantics of EVM. The first one was defined in Lem by Yoichi Hirai (Hirai, 2017). This semantics was defined to prove safety and security properties on specific contracts. It is partially executable and can be used to export Isabelle/HOL theories. The objective here was to compile EVM bytecode to Isabelle/HOL theories so that properties on those *specific* contracts can be proved in Isabelle/HOL. This semantics is very precise w.r.t. specification of low level operations of EVM but it does not precisely follow the gas consumption during calls (see Section 6.2 of (Hirai, 2017)). Thus, this mechanized semantics is not usable, as is, for the proof we want to carry out. Another mechanized semantics is the one by Everett Hildenbrandt et al. (Hildenbrandt et al., 2018) in the K framework. This semantics is fully executable and passes official test suite of EVM (ETS, 2015). This semantics consumes gas at the call point (see `rule <k> callWithCode` in `https://github.com/kframework/evm-semantics/blob/master/evm.md`). In Section 6, we will discuss termination of contracts in this specific setting.

A contract running *out of gas* stops without completing its task and becomes useless. Thus estimating gas consumption of contracts is an active research subject. For instance, (Grech et al., 2018) proposes a static analysis of contract's code to detect resumable loops, loops bounded by inputs, etc. that can lead to an execution running out of gas. Our objective here is different since we aim at proving that whatever the contract code, it cannot loop forever while not spending gas.

2 Ethereum

The blockchain of Ethereum describes the global state of the system, noted σ. In Ethereum a global state σ contains accounts. An *account* is a structure composed of 4 elements: a nonce, a balance (an amount of money in the virtual currency called Ether), a data storage and a code. In Ethereum, there exists two types of accounts: external accounts with an empty code and contracts with a non-empty code.

Calling a contract External accounts are used to store information and Ether. Like in Bitcoin, it is possible to transfer Ether from an account to another through a *transaction*. When a transaction is sent to an account having a code, i.e. a contract, a part of the money is used to pay for the execution of the code[3]. This is called *calling a contract*. When calling a contract, the sent money is not collected by the contract itself but by the miner who accepts to execute contract's code and to add the updated accounts and blocks to the blockchain. In other words, from a given global state σ, the miners produce the new global state σ' resulting of the transactions (and contracts) application on σ. Since adding blocks to the blockchain costs computation power, the miner needs a way to estimate if the reward (money sent to the contract) is competitive with its own computational effort. In Ethereum, this estimation is made possible through the *gas* mechanism. Every basic instruction of contract's code has a fixed cost in gas and every contract claims an (estimated) maximal cost in gas to run its code. Besides, when an account calls a contract it also fixes a

[3]In addition, it is possible to transfer money to a contract, but this part is not important for our termination proof and will not be modeled here.

gas price in Ether. This is used to motivate miners to execute one particular transaction by increasing the gas price and thus their reward.

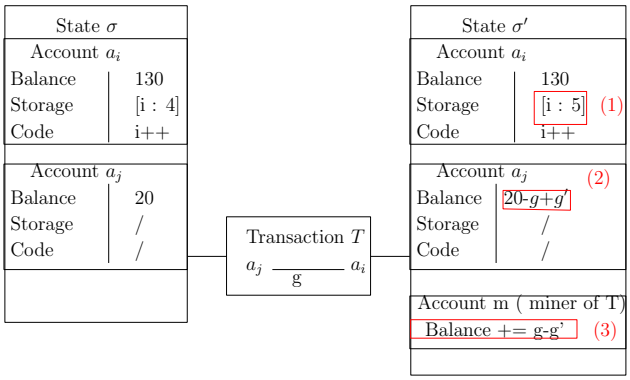

Figure 1: Account a_j calls contract a_i and miner m process the transaction

Example 1. *On the left-hand side of Figure 1, in the state σ, there are two accounts a_i and a_j with a respective balance 130 and 20. Account a_i is a contract and a_j is an external account. Account a_i has a storage called i whose value is* 4. *The code of a_i is simply* `i++`*, i.e., it increments i. Assume that the estimated maximal cost of contract a_i is g. Assume that account a_j builds a transaction T towards a_i, where a_j calls a_i with g gas. To simplify the presentation, we do not consider gas price and assume that one gas costs one Ether. Assume that a miner m processes the transaction T and then adds the new blocks encoding the new values of accounts a_i and a_j in the new blockchain global state σ'. In σ', in the account a_i, i is now* 5 *(1). Note that balance of a_i has not evolved. Balance of a_j has been decreased of g gas unit and increased by g' which is a (possible) gas refund (2). Indeed, contract a_i claims to need g gas units to run its code but less gas may actually be needed. Here we assume that there were g' gas left after the execution of a_i. This gas is refunded to a_j. Finally, the miner m who adds the blocks in σ' is rewarded by $g - g'$ gas (3). Another possibility would have been that execution of a_i needs more than g gas. In this case, the execution of a_i* runs out of gas, *an exception is thrown, the value of i in a_i does not change, the g gas are lost by a_j, and the miner m wins g gas. Precise estimation of gas consumption for contracts is, in itself, a research subject (Grech et al., 2018).*

Creating a contract Any contract c_1 can create a (new) contract c_2 with any arbitrary code, provided that c_1 is given enough gas to store all the instructions of the bytcode of c_2 in the new global state σ'. If contract creation succeeds, this makes contract c_2 publicly available in σ'.

3 Ethereum Virtual Machine: EVM

Contract code is run on the Ethereum Virtual Machine (EVM). Contracts are written in high-level languages such as Solidity (solidity, 2014) or Vyper (vyper, 2017) and compiled to a bytecode format specific to EVM. A bytecode program

is a list of instructions and during the execution a program counter (pc) gives the index of the next instruction to execute. EVM is a stack machine and the effect of arithmetic instructions, test instructions, storage instructions is to read and/or modify this stack, called the execution stack.

There are more than 60 different instructions in EVM. We can split them in 5 families:

Execution stack operations This family encompasses all arithmetic, logic and test instructions like ADD, SUB, AND, OR, EQ, LT, etc. This family also contains instructions that push, pop, swap or duplicate the elements on the execution stack.

Memory access This family contains instructions whose effect is to transfer data between the execution stack and either the temporary local memory (MLOAD, MSTORE) or into the permanent memory (SLOAD, SSTORE). The temporary local memory is a memory where a contract can read and write during its execution and which is erased after contract's completion. The permanent memory is in accounts' storage (thus in the blockchain) and will survive after contract's completion.

Environment operations These are the operations that gather information on the current transaction, the current transaction block and on the current contract. The available pieces of information are: who called this contract, what data was sent to the contract, how many gas unit are left and their price, etc.

Control flow operations Those operations modify the control flow inside the same contract: JUMP, JUMPI (conditional jump), JUMPDEST (marks a jump destination), ...

System operations This family gathers all the operations that permit to create and destroy a contract (CREATE, SUICIDE in (Gavin, 2014), or SELFDESTRUCT in (Gavin, 2019)) and the call and exit operations on contracts (CALL, CALLCODE, DELEGATECALL, RETURN) and additional (REVERT, CALLSTATIC) in (Gavin, 2019).

The differences between the four types of call (CALL, CALLCODE, DELEGATECALL, CALLSTATIC) are subtle. The differences essentially lies in the way the global state is affected by calling the contract and not about the way gas is consumed. The contract called by CALL changes the state of the callee, like in Example 1. The contract called by CALLCODE changes the state of the caller, like when calling a library code. In Example 1, assume that state of account a_j has a field i, then a CALLCODE on a_i, would have incremented the value of this field in the state of account a_j. The DELEGATECALL acts as a CALLCODE except that the identity of the contract caller is different. In a contract c_1, if contract c_2 is called with DELEGATECALL, the call to contract c_2 happens like with a CALLCODE except that identity of the caller is not c_1 but the identity of the caller of c_1. See (Grishchenko et al., 2018a) for details. Finally, CALLSTATIC is similar to CALL except that no modification

of the state is permitted. It can be considered as a "pure" function call without side-effects. However, since there is no difference between the 4 call instructions w.r.t. to gas consumption, we will abstract them in the same way in Section 5.2.

As explained above, to implement the gas mechanism, EVM's designers have chosen to associate each operation with a cost in gas. All operations, except zero-cost operations (STOP, REVERT and RETURN), have a cost strictly greater than zero. Some instructions, like SELFDESTRUCT or SSTORE may result into a gas refund. SELFDESTRUCT destroys the current executed contract and the Ether which may be present in the account is refunded. SSTORE writes information in the permanent storage of the account and, thus, in the blockchain. Refund with SSTORE happens when it replaces a non-zero value by a zero. This kind of erasure permits to save space in the blockchain and is, thus, rewarded. Refunds obtained using SELFDESTRUCT or SSTORE are accumulated during the execution in a *separate* counter and given back after the completion of the whole contract. As a result, during the contract execution, the available gas is not increased by those specific refunds.

Now, to give some intuition about EVM's behavior, we describe more precisely the semantics of some particular instructions. We present all those instructions through their EtherTrust (ethertrust, 2017) small-step semantic rules. The interest of EtherTrust rules w.r.t to the Yellow Paper is that they describe in the same place the effect of the instruction on the state of the system and the gas consumption.

3.1 The ADD instruction

Here is the Yellow Paper semantics (Gavin, 2019) of the ADD operation adding the two elements on top of the execution stack.

ADD	2	1	Addition operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1]$

In this semantics, μ_s is the execution stack and the effect of this operation is to compute a new execution stack μ'_s whose first element $\mu'_s[0]$ is the sum of the two top elements of μ, i.e., $\mu_s[0]$ and $\mu_s[1]$. The cost of the ADD operation is defined in another part of the semantics and is fixed to 3 gas units. Here are the two rules of the EtherTrust small-step semantics of this operation in (ethertrust, 2017).

$$\frac{\iota.code[\mu.\mathsf{pc}] = \mathsf{ADD} \qquad \mu.\mathsf{s} = a :: b :: s \qquad \mu.\mathsf{gas} \geq 3 \qquad \mu' = \mu[\mathsf{s} \rightarrow (a+b) :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 3]}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \rightarrow (\mu', \iota, \sigma, \eta) :: S}$$

$$\frac{\iota.code[\mu.\mathsf{pc}] = \mathsf{ADD} \qquad (|\mu.\mathsf{s}| < 2 \vee \mu.\mathsf{gas} < 3)}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \rightarrow EXC :: S}$$

In this semantics, μ is the local state of the stack machine where $\mu.s$ denotes the execution stack, $\mu.pc$ the program counter, $\mu.gas$ the available gas. The other record ι represents the parameters of the transaction where $\iota.code$ denotes the program under execution. Thus $\iota.code[\mu.pc]$ is the current instruction to execute. Below the line of the semantic rules, $(\mu, \iota, \sigma, \eta) :: S$ is the current call stack. An element of the call stack is called a *frame*, e.g., $(\mu, \iota, \sigma, \eta)$ is the top frame of the current call stack. The field η is a transaction effect where the only information that could be relevant for us w.r.t. gas consumption would be the refund counter. However, as explained in Section 3, this refund counter is separate from the gas available for operation execution. Finally, σ is the

0xf1	CALL	7	1	Message-call into an account. $\mathbf{i} \equiv \boldsymbol{\mu}_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[3] \dots (\boldsymbol{\mu}_\mathbf{s}[3] + \boldsymbol{\mu}_\mathbf{s}[4] - 1)]$ $(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}, I_\mathrm{a}, I_\mathrm{o}, t, t, C_{\mathrm{CALLGAS}}(\boldsymbol{\mu}), & \text{if } \boldsymbol{\mu}_\mathbf{s}[2] \leqslant \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{b} \wedge \\ \quad I_\mathrm{p}, \boldsymbol{\mu}_\mathbf{s}[2], \boldsymbol{\mu}_\mathbf{s}[2], \mathbf{i}, I_\mathrm{e} + 1, I_\mathrm{w}) & \quad I_\mathrm{e} < 1024 \\ (\boldsymbol{\sigma}, g, \varnothing, ()) & \text{otherwise} \end{cases}$ $n \equiv \min(\{\boldsymbol{\mu}_\mathbf{s}[6], \|\mathbf{o}\|\})$ $\boldsymbol{\mu}'_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[5] \dots (\boldsymbol{\mu}_\mathbf{s}[5] + n - 1)] = \mathbf{o}[0 \dots (n-1)]$ $\boldsymbol{\mu}'_\mathbf{o} = \mathbf{o}$ $\boldsymbol{\mu}'_\mathrm{g} \equiv \boldsymbol{\mu}_\mathrm{g} + g'$ $\boldsymbol{\mu}'_\mathbf{s}[0] \equiv x$ $A' \equiv A \uplus A^+$ $t \equiv \boldsymbol{\mu}_\mathbf{s}[1] \mod 2^{160}$ where $x = 0$ if the code execution for this operation failed due to an exceptional halting (or for a REVERT) $\boldsymbol{\sigma}' = \varnothing$ or if $\boldsymbol{\mu}_\mathbf{s}[2] > \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{b}$ (not enough funds) or $I_\mathrm{e} = 1024$ (call depth limit reached); $x = 1$ otherwise. $\boldsymbol{\mu}'_\mathrm{i} \equiv M(M(\boldsymbol{\mu}_\mathrm{i}, \boldsymbol{\mu}_\mathbf{s}[3], \boldsymbol{\mu}_\mathbf{s}[4]), \boldsymbol{\mu}_\mathbf{s}[5], \boldsymbol{\mu}_\mathbf{s}[6])$ Thus the operand order is: gas, to, value, in offset, in size, out offset, out size. $C_{\mathrm{CALL}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv C_{\mathrm{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) + C_{\mathrm{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu})$ $C_{\mathrm{CALLGAS}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} C_{\mathrm{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) + G_{callstipend} & \text{if } \boldsymbol{\mu}_\mathbf{s}[2] \neq 0 \\ C_{\mathrm{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{otherwise} \end{cases}$ $C_{\mathrm{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} \min\{L(\boldsymbol{\mu}_\mathrm{g} - C_{\mathrm{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu})), \boldsymbol{\mu}_\mathbf{s}[0]\} & \text{if } \boldsymbol{\mu}_\mathrm{g} \geq C_{\mathrm{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \\ \boldsymbol{\mu}_\mathbf{s}[0] & \text{otherwise} \end{cases}$ $C_{\mathrm{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv G_{call} + C_{\mathrm{XFER}}(\boldsymbol{\mu}) + C_{\mathrm{NEW}}(\boldsymbol{\sigma}, \boldsymbol{\mu})$ $C_{\mathrm{XFER}}(\boldsymbol{\mu}) \equiv \begin{cases} G_{callvalue} & \text{if } \boldsymbol{\mu}_\mathbf{s}[2] \neq 0 \\ 0 & \text{otherwise} \end{cases}$ $C_{\mathrm{NEW}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} G_{newaccount} & \text{if } \mathtt{DEAD}(\boldsymbol{\sigma}, \boldsymbol{\mu}_\mathbf{s}[1] \mod 2^{160}) \wedge \boldsymbol{\mu}_\mathbf{s}[2] \neq 0 \\ 0 & \text{otherwise} \end{cases}$

Figure 2: The Yellow Paper definition of the CALL operation.

current state of the global state. Since, there are no side effects, every update on this global state is propagated by the semantic rules. In the first rule, for ADD, there is enough gas to execute ADD and an execution stack with at least two elements. Thus, the call stack becomes $(\mu', \iota, \sigma, \eta) :: S$ where μ' is μ with an updated execution stack, an increased program counter $\mu.pc$, and a $\mu.gas$ decreased of 3 gas units. The second rule defines the execution of ADD when there are not enough elements on the execution stack or not enough gas to execute ADD. This results into stacking an exception frame (EXC) on top of the call stack.

3.2 The CALL instruction

Yellow Paper's definition for the CALL operation is given in Figure 2. In this definition, $\mu'.g$ is the gas after the execution of this instruction. The value of $\mu'.g$ is set to $\mu.g+g'$ where g' is the gas remaning after the execution of the called contract (gas refund). In fact, the execution of the CALL instruction itself has a cost which is subtracted from $\mu'.g$ (rule (135) of the Yellow Paper semantics) for executing the instruction. Thus $\mu'.g$ should be read as $\mu.g - CallCost + g'$. Fortunately, EtherTrust provides a higher level and self-contained small-step interpretation of the semantics of this operation. In the following, since they are more readable, we only present the EtherTrust version of EVM semantic rules.

$$\frac{\begin{array}{c}\iota.code\,[\mu.\mathsf{pc}] = \mathsf{CALL} \qquad \mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s \\ \sigma(to) \neq \bot \qquad |A| + 1 < 1024 \qquad \sigma(\iota.\mathsf{actor}).\mathsf{b} \geq va \qquad aw = M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(va, 1, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(va, 1) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ \mu.\mathsf{gas} \geq c \qquad \sigma' = \sigma\langle to \to \sigma(to)[\mathsf{b} \mathrel{+}= va]\rangle\langle \iota.\mathsf{actor} \to \sigma(\iota.\mathsf{actor})[\mathsf{b} \mathrel{-}= va]\rangle \\ d = \mu.\mathsf{m}\,[io, io + is - 1] \qquad \mu' = (c_{call}, 0, \lambda x.\,0, 0, \epsilon) \\ \iota' = \iota[\mathsf{sender} \to \iota.\mathsf{actor}][\mathsf{actor} \to to][\mathsf{value} \to va][\mathsf{input} \to d][\mathsf{code} \to \sigma(to).\mathsf{code}]\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to (\mu', \iota', \sigma', \eta) :: (\mu, \iota, \sigma, \eta) :: S}$$

This small-step semantic rule defines the CALL execution when everything is OK: the execution stack contains enough arguments to perform the call ($\mu.s$ has at least 7 elements), there is enough gas to perform the call $\mu.gas \geq c$, and there is room in the call stack to add a new frame ($|A| + 1 < 1024$). The cost c is the sum of the costs for calling the CALL instruction itself (700 gas units) plus a variable cost depending on the size of the input and output of the contract: this gas is paid when reading contract parameters and outputing its future result. On the lower part of this rule, the call stack $(\mu, \iota, \sigma, \eta) :: S$ becomes $(\mu', \iota', \sigma', \eta') :: (\mu, \iota, \sigma, \eta) :: S$ where $(\mu', \iota', \sigma', \eta')$ is a new frame stack which has been added on top of the call stack, where μ' is a new record, where $\mu'.gas = c_{call}$ is the gas transferred to the new frame stack by the old one and $\mu'.pc$ is set to 0. The code to execute in this new frame is $\iota'.code = \sigma(to).code$ where $\sigma(to)$ is the account receiving the call. Note that, like it was stated in the above sections, the new call stack is $(\mu', \iota', \sigma', \eta') :: (\mu, \iota, \sigma, \eta) :: S$ where the gas sent to the new frame ($\mu'.gas$) has not been subtracted from the frame $(\mu, \iota, \sigma, \eta)$ (μ is the same, thus so is $\mu.gas$). The gas is retracted when the contracts returns. Note also that this is compatible with the Yellow Paper semantics where, to update the gas w.r.t. the execution of the CALL, one has to know how much gas g' will be refunded *after* the execution of the called contract.

3.3 The RETURN instruction

Contract returning is performed by two rules. The first one processes the RETURN operation.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{RETURN} \\ \mu.\mathsf{s} = io :: is :: s \qquad aw = M\,(\mu.\mathsf{i}, io, is) \qquad c = C_{mem}\,(\mu.\mathsf{i}, aw) \\ valid\,(\mu.\mathsf{gas}, c, |s|) \qquad d = \mu.\mathsf{m}[io, io + is + 1] \qquad g = \mu.\mathsf{gas} - c\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\to \mathit{HALT}(\sigma, g, d, \eta) :: S}$$

In this rule the current instruction to execute $\iota.code[\mu.pc]$ is abbreviated by $\omega_{\mu,\iota}$. The effect of this rule is to replace the frame by an HALT frame with the information that should be provided to the caller, i.e., the possible updates on the global state σ, the remaining gas g, a result d and transaction effects η. Finally, the HALT frame is popped by a second rule.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CALL} \\ \mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s \qquad to_a = to \mod 2^{160} \\ flag = \sigma.to_a = \bot\,?\,0 : 1 \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(va, flag, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(va, flag) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ \mu' = \mu[\mathsf{i} \to aw][\mathsf{s} \to 1 :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{+}= gas - c][\mathsf{m} \to \mu.\mathsf{m}[[oo, oo + s - 1] \to d]]\end{array}}{\Gamma \vDash \mathit{HALT}(\sigma', \eta', gas, d) :: (\mu, \iota, \sigma, \eta) :: S \;\to (\mu', \iota, \sigma', \eta') :: S}$$

This is the rule for the standard case, i.e., in the frame below the HALT frame, the current instruction is a CALL and the execution stack contains all the information that were necessary to perform the call. Then, we retract the gas units necessary to perform the call (noted c) and refund gas units of gas coming from the HALT frame. The global store σ' coming from the HALT frame replaces σ in the current frame.

3.4 The CREATE instruction

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{CREATE} \\ \mu.\mathsf{s} = va :: io :: is :: s \qquad aw = M\,(\mu.\mathsf{i}, io, is) \qquad c = C_{mem}\,(\mu.\mathsf{i}, aw) + 32000 \\ valid\,(\mu.\mathsf{gas}, c, |s|+1) \qquad va \leq \sigma(\iota.\mathsf{actor}).\mathsf{balance} \\ |S|+1 \leq 1024 \qquad \rho = newAddress\,(\iota.\mathsf{actor}, \sigma(\iota.\mathsf{actor}).\mathsf{nonce}) \\ \sigma(\rho) \neq \bot \qquad b = \sigma(\rho).\mathsf{balance} + va \\ \sigma' = \sigma\langle\rho \to (0, b, \lambda x.\,0, \epsilon)\rangle\langle\iota.\mathsf{actor} \to \sigma(\iota.\mathsf{actor})[\mathsf{balance} \mathrel{-}= va][\mathsf{nonce} \mathrel{+}= 1]\rangle \\ i = \mu.\mathsf{m}\,[io, io + is - 1] \\ \iota' = \iota[\mathsf{sender} \to \iota.\mathsf{actor}][\mathsf{actor} \to \rho][\mathsf{value} \to va][\mathsf{code} \to i][\mathsf{input} \to \epsilon] \\ \mu' = (L\,(\mu.\mathsf{gas} - c), 0, \lambda x.\,0, 0, \epsilon)\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \to (\mu', \iota', \sigma', \eta) :: (\mu, \iota, \sigma, \eta) :: S}$$

This rule gives the contract creation. The effect of the create is to stack a new frame $(\mu', \iota', \sigma', \eta)$ on top of the call stack, which corresponds to the new contract to create and to execute. The code associated to this contract $\iota'[code]$ is $i = \mu.m[io, io + is - 1]$ comes from the local memory of the contract which executes the CREATE operation. One can remark that this rule is very close to a CALL. The only difference is that the code $\iota'[code]$ is not obtained from the global store σ but from the memory of the current contract. This code will be later attached to the address of the new contract in the global state of the system, when the topmost frame becomes a HALT frame, by another semantic rule (see (ethertrust, 2017) for details).

4 An EVM model specialized for gas analysis

The gas mechanism ensures that a contract can only run a finite number of "local" instructions, i.e., instructions whose effect is local to the current contract (no call, return, etc.). As mentioned above, when a contract c_1 calls another contract c_2 with, say, g units of gas, the gas associated to c_1 *is not charged immediately*. Thus, using Yellow Paper semantics, a contract c_1 calling itself is indefinitely looping. The Yellow Paper prevents this by fixing a call stack size limit. If a contract exhausts the stack limit then its execution ends by an exception. However, unlike other virtual machines, EVM has no exception catching mechanism. When an exception is raised in a contract c, the execution of c stops, the information of the contract c is popped from the stack and the control flow goes back to the previous contract in the stack if it exists, otherwise the execution stops.

To sum up, termination of contracts in the formal semantics of the Yellow Paper is enforced by the gas mechanism and the fact that the call stack is finite. In the following, to formally prove termination we prove that, whatever the contracts may be, the call stacks decrease w.r.t. a well founded-ordering. First, we define the call stacks and the frames composing the call stacks based on the formal small-step semantics of (Gavin, 2014; Gavin, 2019) and (Grishchenko et al., 2018b; Grishchenko et al., 2018a).

The maximal call stack size The maximal call stack size is denoted by *stack_lim*. We assume that *stack_lim* is a natural number strictly greater to 0.

Abstraction of the frames For running a contract c_1, the EVM stores information in the call stack. In the following, we call this information a *frame*. Following (Grishchenko et al., 2018b), our frames can denote standard program execution frames, HALT frames and EXC frames. In our EVM model specialized

for gas analysis, we can abstract frames by three different frame forms: either $Ok(g, pc, p, e)$, $Halt(g, e)$ or $Exception$, where g is a gas value, pc is a program counter, p is a program code, and e is an environment. Like in (Grishchenko et al., 2018b), this environment is an abstraction of the global state of the system σ. In our model, this environment maps contract names to the associated codes. An $Ok(g, pc, p, e)$ frame represents a standard execution frame $(\mu, \iota, \sigma, \eta)$, where we abstract away η and most parts of μ (including the execution stack and the local memory). In μ, we only keep track of $\mu.pc$ the program counter and $\mu.gas$ the available gas. Similarly, we forget everything about ι except $\iota.code$ the current program to execute. In σ, we only follow the contract names associated to code and forget about all other type of information. A $Halt(g, e)$ frame represents a contract that successfully reaches a RETURN instruction, where g is the gas remaining after the execution of the contract (the refund) and e is the (possibly) modified environment. In particular, e may contain new contract names and their associated code. On the opposite, the result value d and the effect η are not stored in our abstract version of the semantics, because they have no impact on the control flow nor on gas consumption. In particular, if a conditional jump depends on the result d then this will be modelled in our abstract semantics by the fact that the abstract `Jump` instruction can jump to any valid position in the current contract. Finally an $Exception$ frame represents a contract whose execution has failed because it exhausted the available gas, overflowed the call stack, jumped to an invalid pc or tried to execute an undefined instruction.

The call stacks Call stacks will be represented by lists of frames, where the top of the stack is the left-most element of the list.

Example 2. *(1) Assume that we are running a unique contract c_1 having 18 gas units left, a program counter pc, a program p and an environment e. The corresponding call stack will thus be $[Ok(18, pc, p, e)]$. (2) Assume that the instruction at position pc in p is a CALL to contract c_2 with a calling gas value of 10, then the call stack becomes $[Ok(10, 0, p2, e2), Ok(18, pc, p, e)]$, where $p2$ and $e2$ are the program and environment associated to c_2. (3) Now assume that the instruction at position 0 in $p2$ consumes 2 gas units, the call stack is now $[Ok(8, 1, p2, e3), Ok(18, pc, p, e)]$ where $e2$ may have been transformed into $e3$. (4) Then, assume that contract c_2 reaches program point $pc2$ with 4 gas units left and the environment $e4$: $[Ok(4, pc2, p2, e4), Ok(18, pc, p, e)]$. (5) At $pc2$ in $p2$ there is a RETURN instruction so that c_2 halts on a valid state. The call stack becomes: $[Halt(4, e4), Ok(18, pc, p, e)]$. (6) Then, the frame of contract c_2 is popped and control is returned back to c_1 that called c_2. When returning back to c_1, we have to consume all the gas used for the call: the cost of the call instruction itself with the cost of calling c_2. Assume that the call instruction costs 3 gas. Thus, we need to consume 3 gas plus the gas that was planned at step (2) for calling contract c_2: 10. Besides, we refund the 4 gas returned by Halt and place the environment $e4$ into c_1 frame. Thus, the call stack becomes $[Ok(9, pc + 1, p, e4)]$. (7) Now we assume that, the execution of contract c_1 ends with an exceptional state. The resulting stack is thus $[Exception]$.*

5 Termination proof for the Yellow Paper semantics in Isabelle/Hol

5.1 The Termination measure

A usual technique to prove termination of a recursive function f mapping values of type A to values of type B is to define a well-founded strict ordering $\succ$ on elements of type A. This ordering has to be defined such that for all $x \in A$ if $f(x)$ evaluates to $f(y)$, noted $f(x) \rightsquigarrow f(y)$, then we have $x \succ y$. If such a well-founded ordering $\succ$ exists then it proves termination of f. Indeed, for any infinite derivation $f(t_1) \rightsquigarrow f(t_2) \rightsquigarrow \ldots$, we have an infinite chain $t_1 \succ t_2 \succ \ldots$, which contradicts the fact that $\succ$ is well-founded.

To prove termination of the EVM semantics, we need to show that when executing one EVM bytecode on a stack s_1 we obtain a stack s_2 which is strictly smaller to s_1 w.r.t. a well-founded ordering $\succ$. Finding such an order is not straightforward as we show on the following example. For instance, to prove termination on the execution of Example 2, we need a well-founded ordering $\succ$ such that

$$
\begin{array}{lrl}
(1) & [Ok(18, pc, p, e)] & \succ \\
(2) & [Ok(10, 0, p2, e2), Ok(18, pc, p, e)] & \succ \\
(3) & [Ok(8, 1, p2, e3), Ok(18, pc, p, e)] & \succ \\
(4) & [Ok(4, pc2, p2, e4), Ok(18, pc, p, e)] & \succ \\
(5) & [Halt(4, e4), Ok(18, pc, p, e)] & \succ \\
(6) & [Ok(9, pc + 1, p, e4)] & \succ \\
(7) & [Exception] &
\end{array}
$$

Since we may have loops in a frame, we may have two consecutive frames with the same pc or ascending pc. Thus, the program counter is not relevant for the ordering. In the same way, since environments $e, e2, e3, e4$ and programs $p, p2$ may not evolve between two frames, they are hardly usable for a strict ordering. Hence, the ordering can only depend on the gas value of the frames. If we abstract away anything but gas from the previous example, we obtain:

$$
\begin{array}{rl}
[Ok(18)] & \succ \\
[Ok(10), Ok(18)] & \succ \\
[Ok(8), Ok(18)] & \succ \\
[Ok(4), Ok(18)] & \succ \\
[Halt(4), Ok(18)] & \succ \\
[Ok(9)] & \succ \\
[Exception] &
\end{array}
$$

Note that, using a simple ordering for $\succ$ does not satisfy the above ordering chain. For instance, the following orderings fail:

- comparing the size of the list:
  $[Ok(18)] \not\succ [Ok(10), Ok(18)]$
- comparing the gas value of the topmost frame:
  $[Halt(4), Ok(18)] \not\succ [Ok(9)]$

- comparing the gas value of frames from bottom to top:
  $[Ok(18)] \not\succ [Ok(10), Ok(18)]$
- comparing the sum of the gas values:
  $[Ok(18)] \not\succ [Ok(10), Ok(18)]$
- or, lexicographic combinations of them starting from the leftmost part of the list:
  $[Halt(4), Ok(18)] \not\succ [Ok(9)]$
- or, lexicographic combinations of them starting from the rightmost part of the list:
  $[Ok(18)] \not\succ [Ok(10), Ok(18)]$

The order we define to prove termination of EVM semantics is based on *measure* functions, i.e., functions mapping frames to natural numbers. Thus, stacks can be evaluated into lists of natural numbers and lists of natural numbers are compared using a lexicographic combination of the order $>$ on natural numbers. Before defining our measure functions, we complete the call stacks by dummy frames up to the frame stack's maximal size $stack\_lim$. These dummy frames (noted D) have a gas value depending on the type of the topmost frame and on its gas value if there is one (for Ok and $Halt$) and 0 otherwise (for $Exception$). If the topmost frame is $Ok(i)$ then the dummy frames will be $D(i+3)$, if the topmost frame is $Halt(i)$ then the dummy frames will be $D(i+2)$. If the topmost frame is $Exception$ then the dummy frames will be $D(0)$. Assuming that the maximal stack size is 4, the frame stacks of our previous example will be completed up to size 4 in the following way:

$$\begin{array}{llllll}
[& D(21), & D(21), & D(21), & Ok(18) &] \succ \\
[& D(13), & D(13), & Ok(10), & Ok(18) &] \succ \\
[& D(11), & D(11), & Ok(8), & Ok(18) &] \succ \\
[& D(7), & D(7), & Ok(4), & Ok(18) &] \succ \\
[& D(6), & D(6), & Halt(4), & Ok(18) &] \succ \\
[& D(12), & D(12), & D(12), & Ok(9) &] \succ \\
[& D(0), & D(0), & D(0), & Exception &]
\end{array}$$

Using this completion of call stacks, the order becomes straightforward: we compare frame's measures lexicographically, starting from the rightmost part of the list, i.e., from the bottom of the stack. We use the following measure function for frames: $measure(Ok(i)) = i + 3$, $measure(Halt(i)) = i + 2$, $measure(D(i)) = i$ and $measure(Exception) = 1$. Thus, on the above example, we have $[D(21), D(21), D(21), Ok(18)] \succ [D(13), D(13), Ok(10), Ok(18)]$ because the 4^{th} element of the two stacks are equal ($Ok(18)$) but the 3^{rd} element of the first stack has a measure of 21 where the 3^{rd} element of the second stack has a measure of 13. Similarly, we have $[D(21), D(21), Ok(4), Ok(18)] \succ [D(13), D(13), Halt(4), Ok(18)]$ because $measure(Ok(4)) = 7$ and $measure(Halt(4)) = 6$.

The values for the measure of frames, $measure(Ok(i)) = i+3$, $measure(Halt(i)) = i + 2$, $measure(D(i)) = i$ and $measure(Exception) = 1$, have been chosen so that an Ok frame halting (correctly) with a gas i and moving to a $Halt$ with the same gas value i can be ordered. With this measure, we have $[Ok(i), f_1, ..., f_n] \succ$

$[Halt(i), f_1, ..., f_n]$, for all $i \geq 0$ and all frames $f_1, \ldots, f_n$. This is crucial since this sequence of frame stacks is possible with the EVM semantics.

Definition 1 (Stack measure)**.** *Let Es be the maximal height of the EVM call stack. Let s be an EVM call stack represented by a list of frames of the form $Ok(i)$, $Halt(j)$, or $Exception$ where i, j are strictly positive natural numbers. Let $s(k)$ be the k-th element of the stack s for $0 \leq k < |s|$, thus $s(0)$ is the topmost frame. For $0 \leq k < |s|$, let*

$$m_k = \begin{cases} i+3 \text{ if } s(k) = Ok(i) \\ i+2 \text{ if } s(k) = Halt(i) \\ \mathbf{1} \text{ if } s(k) = Exception \end{cases}$$

$$d = \begin{cases} i+3 \text{ if } s(0) = Ok(i) \\ i+2 \text{ if } s(0) = Halt(i) \\ \mathbf{0} \text{ if } s(0) = Exception \end{cases}$$

The stack measure of s is a list of natural numbers, of length Es, defined by:

$$measure(s) = \underbrace{[d, \ldots, d]}_{Es-|s|} @ [m_0, \ldots, m_{|s|-1}]$$

where @ denotes list concatenation.

With this measure, we can prove the following termination theorem.

Theorem 1. *The execution of any contract on the EVM terminates.*

The proof amounts to showing that each EVM execution step results in a decrease of the measure on call stacks defined in Definition 1. To prove this formally, we need to define an abstract version of the EVM semantics specialized for gas analysis. This will be done in the next section where we propose an Isabelle/HOL formalization of this specialized semantics.

5.2 Implementation in Isabelle/HOL

The Isabelle datatype for instructions and the type for programs are the following

```
type_synonym
  gas = nat

type_synonym
  pc= nat

datatype
  instr = Nil
  | Local "gas"
  | Jump "gas"
  | Call "gas*gas*contractName"
  | Stop

type_synonym program = "instr list"
```

The abstraction of frames defined in Section 5.1 only keeps track of the gas, the current program to execute, the program counter, and the environment.

With this abstraction, many EVM instructions have a similar behavior and can be abstracted by a general `Local` instruction whose only parameter is its gas cost. The `Local(g)` instruction represents any instruction whose effect is local to the current frame, does not affect the control flow, and whose cost in gas is the natural number `g`. This instruction represents all instructions of the families **Execution stack operations**, **Memory access** and **Environment operations** of Section 3, i.e., instructions such as ADD, SSTORE, MSTORE, LT, AND, PUSHi, POP, DUPi, SWAPi, ... The `Nil` instruction stands for undefined instruction (an undefined opcode) that may appear in a program or the INVALID instruction. The `Jump(g)` instruction represents the JUMP and JUMPI instructions where `g` is the cost of executing the jump. There is no destination associated with the `Jump` instruction because the abstract semantics will arbitrarily chose the destination when executing the `Jump`. This is an over-approximation of all the possible JUMP and JUMPI behaviors with any position in the contract tagged by a JUMPDEST instruction. Thus, we cover all the instructions of the **Control flow operations** family of Section 3. The family of **System operations** is represented by two different abstract instructions. The `(Call gcall ccall cname)` instruction represent EVM's CREATE, CALL, CALLCODE, DELEGATECALL, and CALLSTATIC where `gcall` is the cost in gas of executing the call instruction itself, `ccall` is the gas transferred to the called contract, and `cname` is the contract name to be called. Having CREATE and CALL abstracted by the same `Call` instruction is coherent with EVM semantics, where the difference between the two is small. In the case of a CALL, the contract name exists in the environment and is associated to a program. In the case of a CREATE the contract name does not exist and the association is added in the environment of the new frame. Those two possible behaviors are defined in our semantics (see Section 5.6). The last abstract instruction for the family **System operations** is the `Stop` instruction which represents STOP, RETURN, REVERT and SELFDESTRUCT EVM's instructions. Finally, a program p is a list of such instructions and a program counter pc of p is a position between 0 and $length(p) - 1$ in this list.

Note that, in EVM, all instructions (except STOP, REVERT and RETURN) have a gas cost which is strictly greater than zero. However, the above Isabelle/HOL datatype only imposes that gas costs are of type `nat`, i.e., that they are greater *or equal* to zero. Thus, we complement this datatype with a `valid_prog(p)` predicate stating that, in a program `p`, every instruction with a cost g is such that $g > 0$. A program is valid if it satisfies this predicate. As explained above, frames can contain different pieces of information: programs, program counter, gas value and an environment. In EVM, environments contain different types of values for variables. In our gas-oriented model, we focus on environments (type `env` in the following) mapping contract names (i.e. strings) to programs. Thus, we also define a predicate `valid_env` ensuring that an environment maps contract names to valid programs.

The function defining the EVM semantics is `smallstep` and its Isabelle/HOL type is `call_stack` $\Rightarrow$ `call_stack`. Starting from a call stack, whose top frame is $Ok(g, pc, p, e)$ this function executes the instruction at position pc in p with environment e and returns the resulting call stack. Recall that there are three kinds of frames: *Ok*, *Halt* or *Exception*. The Isabelle/HOL type `call_stack` is simply a list of frames. Thus, this type includes invalid call stacks, i.e., stacks that contain frames whose program is invalid, and stacks

that cannot be produced by a correct execution of the EVM semantics (such as $[Exception, Exception]$). Since functions in Isabelle/HOL have to be total, we need to define the `smallstep` function for all stacks including the invalid ones. To ensure totality of `smallstep`, while preserving its soundness w.r.t. EVM, we map any invalid call stacks to the result stack $[Invalid\_frame]$, where $Invalid\_frame$ is an additional kind of frame. Here is the corresponding Isabelle/HOL datatype.

```
datatype
  frame =
    Ok "gas*pc*program*env"
    | Exception
    | Halt "gas*env"
    | Invalid_frame

type_synonym
  call_stack = "frame list"
```

We define a predicate `valid_stack` checking that a call stack is valid: it contains only valid programs, valid environments and valid sequence of frames. A valid sequence does not contain $Invalid\_frame$, and $Exception$ or $Halt$ cannot be below other frames. We now present the `smallstep` function of type `smallstep::"call_stack ⇒ call_stack"` and whose role is to execute the abstract instructions on a call stack. The complete Isabelle/HOL code can be found here (EFSyellow, 2020). Note that this semantics is executable and some examples can be found and run at the end of the theory file. We here only give some excerpts of the `smallstep` function.

5.3 Semantics for Stop, Nil and Local instructions

The first one illustrates the execution of `Stop`, `Nil` and `Local` instructions. Recall that the `Local` instruction covers the **Execution stack**, **Memory access** and **Environment** families of operations of Section 3. This code has to be compared with the semantic rules of Section 3.1.

```
"smallstep ((Ok (g,pc,p,e))#l) =
   (case p.(pc) of
      Stop ⇒ ((Halt (g,e))#l) |
      Nil  ⇒  (Exception#l)|
      Local(n) ⇒(if (n>0) then (
                  if (n≤g) then
                    ((Ok ((g-n),pc+1,p,e))#l)
                  else (Exception#l))
                else [Invalid_frame] )|
[...]
```

In the case of a `Local(n)` instruction, if $n = 0$ this results into a $[Invalid\_frame]$. Otherwise if n is lesser or equal to the available gas g then instruction is executed, gas is updated and pc is incremented. Otherwise, an exception is stacked on the call stack.

5.4 Semantics for the Jump instuction

Now, we present the semantics of the `Jump` instruction which covers the operations of the **Control flow** family of Section 3.

```
"smallstep ((Ok (g,pc,p,e))#l) =
  (case p.(pc) of
  [...]
     Jump(n) ⇒
       if (n>0) then
         (let pj= (any_jump 0) in
            if (n≤g) then
              (if (pj<(length p)) then
                 ((Ok(g-n,pj,p,e))#l)
               else (Exception#l))
            else (Exception#l))
       else [Invalid_frame] )
  [...]
```

Like above, for `Local(n)` if $n = 0$ this results into a $[Invalid\_frame]$. Otherwise we compute an arbitrary value for the destination of the jump, named `pj`, using the function `any_jump`. This function is left unspecified, we only known its type `any_jump::'a ⇒ nat`. Thus, `pj= (any_jump 0)` associates any natural number to `pj`. This models the fact that the jump can be conditional and JUMPDEST labels can be attached to any part of the current contract. Then, if there is enough gas to execute the jump `(n≤g)` and the jump destination is in the range of the current contract `(pj<(length p))` then the program counter is updated with `pj` and the top frame becomes `(Ok(g-n,pj,p,e)`. Otherwise, an exception is stacked on the call stack.

5.5 Semantics for the CALL return

The semantics of the contract call is straightforward, see (EFSyellow, 2020). Thus, the third excerpt, illustrates the return of a contract call. This has to be compared with the second rule of Section 3.3.

```
[...]
"smallstep ((Halt(gend,ef))#(Ok(g,pc,p,e))#l) =
  (case p.(pc) of
    Call(gcall,ccall,name) ⇒
      if ((gcall+ccall)>g) then [Invalid_frame]
      else if (gcall≤0) then [Invalid_frame]
      else if (ccall≤0) then [Invalid_frame]
      else
        if (ccall<gend) then
          (Exception#(Ok (g,pc,p,e))#l)
        else
          ((Ok((g+gend-gcall-ccall),
               (pc+1),p,ef))#l)
  |_ ⇒ [Invalid_frame] )"|
[...]
```

When a contract halts correctly (frame `(Halt (gend,ef))` on top of the stack, with gas refund `gend` and environment `ef`) then if the frame below is a frame `(Ok (g,pc,p,e))` such that the instruction at position `pc` in `p` is a call,

and such that all calling conditions were satisfied before the call, then we pop the *Halt* frame and continue in the *Ok* frame, with gas `(g+gend-gcall-ccall)`, at position `pc+1` and with (possibly) modified environment `ef`. Any other behavior results into an [*Invalid_frame*].

5.6 Semantic for the CREATE instruction

Finally, here is an excerpt illustrating the CREATE. This has to be compared with the rule Section 3.4.

```
"smallstep ((Ok (g,pc,p,e))#l) =
  (case p.(pc) of
  [...]
    Call (gcall,ccall,name) ⇒
      if ((gcall>0)∧(ccall>0)∧
          (((length l)+1)<stack_lim)∧
           ((gcall+ccall)≤g)) then
        (case e(name) of
          None ⇒
           (let pnew=(any_valid_program 0) in
             ((Ok (ccall,0,pnew,
                   e( name := (Some pnew)))
               )#(Ok(g,pc,p,e))#l))
    [...]
```

The CREATE is simulated by a CALL where the contract name `name` is undefined in the environment. In this case, we create an arbitrary program using the `any_valid_program` function of type `any_valid_program::'a ⇒ program`. This function is left unspecified but we assume in the Isabelle theory that its result is always a valid program, i.e., a program whose all gas costs are strictly positive.

5.7 Soundness and termination proof

Since we completed the EVM semantics with a new type of frames (*Invalid_frame*) to have a total function `smallstep`, we first need to verify that this modification does not break the EVM semantics encoded in the `smallstep` function. This can be checked using the following Isabelle/HOL theorem stating that validity of stacks is preserved by `smallstep`.

```
lemma validstack_smallstep:
  "(valid_stack l)⟶
         (valid_stack (smallstep l))"
```

In other words, when running `smallstep` on a valid stack, then *Invalid_frame* will never show up. The (complete) execution of a contract starts from a call stack, applies the `smallstep` function until a `Halt`, `Exception` or `Invalid_frame` is reached. The result of a complete execution is a single frame. It is defined in Isabelle/HOL in the `execute` function as follows:

```
function (sequential)
          execute :: "call_stack ⇒ frame"
where
"execute ([]) = Invalid_frame"|
```

```
"execute ([Halt (g,e)]) = (Halt (g,e))"|
"execute ([Exception]) = (Exception)"|
"execute ([Invalid_frame]) = Invalid_frame"|
"execute l = (if (length l > stack_lim) then
                Invalid_frame
              else execute (smallstep l))"
```

Again, we can lift the previous theorem to prove that adding $Invalid\_frame$ does not break the semantics, i.e. executing a valid stack always result into a valid stack, where $stack\_lim$ is an arbitrary constant (greater than 0) which defines the maximal stack size.

```
lemma finalSoundnessTheorem:
 "(valid_stack l ∧ (length l ≤ stack_lim))
     ⟶ (valid_stack [(execute l)])"
```

Now, we can state and prove in Isabelle/HOL the termination theorem (Theorem 1) which corresponds to the termination proof of the `execute` function. The proof of this property relies on the measure technique described in section 5.1 extended with $measure$(`Invalid_frame`) = 1 and encoded into Isabelle/HOL. Note that this final termination theorem is valid for any stack size ($stack\_lim$), where the termination measure is the one defined in Section 5.1 and formalized by the `list_order` Isabelle/HOL function.

```
termination execute
  apply (relation
          "measures (list_order stack_lim)")
[...]
```

The Isabelle/HOL development is around 1200 lines. Excluding definitions, the proof of soundness is composed of 18 intermediate lemmas and of 300 lines of Isabelle/HOL. The proof of termination is composed of 57 intermediate lemmas and of 400 lines of Isabelle/HOL.

6 Termination proof for the EVM reference implementations semantics

As explained in the introduction, implementations generally use a slightly different semantics for the call: g is retracted to c_1 at the calling point for c_2 and g_{refund} is added when the control flow returns from c_2. This is the case for (pevm, 2017) (see `class BaseCall` and `class Call(BaseCall)` in `https://github.com/ethereum/py-evm/blob/master/eth/vm/logic/call.py`). Executing Example 2 with this other semantics yields the following sequence of call stacks.

Example 3. *(1) Assume that we are running a unique contract c_1 having* 18 *gas units left, a program counter pc, a program p and an environment e. The corresponding call stack will thus be $[Ok(18, pc, p, e)]$. (2) Assume that the instruction at position pc in p is a CALL to contract c_2 with a calling gas value of* 10*, and the cost of a CALL is 3. Then the call stack becomes*

$$[Ok(10, 0, p2, e2), Ok(5, pc, p, e)],$$

where $p2$ *and* $e2$ *are the program and environment associated to* c_2*. (3) Now assume that the instruction at position* 0 *in* $p2$ *consumes 2 gas units, the call stack is now*

$$[Ok(8, 1, p2, e3), Ok(5, pc, p, e)]$$

where $e2$ *may have been transformed into* $e3$*. (4) Then, assume that contract* c_2 *reaches program point* $pc2$ *with* 4 *gas units left and the environment* $e4$*:*

$$[Ok(4, pc2, p2, e4), Ok(5, pc, p, e)].$$

(5) At $pc2$ *in* $p2$ *there is a RETURN instruction so that* c_2 *halts on a valid state. The call stack becomes:*

$$[Halt(4, e4), Ok(5, pc, p, e)].$$

(6) Then, the frame of contract c_2 *is popped and control is returned back to* c_1 *that called* c_2*, the* 4 *gas are refunded to* c_1 *and we place the environment* $e4$ *into* c_1 *frame. Thus, the call stack becomes*

$$[Ok(9, pc + 1, p, e4)].$$

(7) Now we assume that, the execution of contract c_1 *ends with an exceptional state. The resulting stack is thus* $[Exception]$*.*

To prove termination we now need a well-founded strict ordering that satisfy the following ordering constraints:

$$\begin{array}{ll}
(1) & [Ok(18, pc, p, e)] \succ \\
(2) & [Ok(10, 0, p2, e2), Ok(5, pc, p, e)] \succ \\
(3) & [Ok(8, 1, p2, e3), Ok(5, pc, p, e)] \succ \\
(4) & [Ok(4, pc2, p2, e4), Ok(5, pc, p, e)] \succ \\
(5) & [Halt(4, e4), Ok(5, pc, p, e)] \succ \\
(6) & [Ok(9, pc + 1, p, e4)] \succ \\
(7) & [Exception]
\end{array}$$

Note that the ordering used for the previous semantics does not satisfy those constraints. If we complete our stacks up to size 4, we obtain:

$$\begin{array}{rrrrcl}
[& D(21), & D(21), & D(21), & Ok(18) &] \succ \\
[& D(13), & D(13), & Ok(10), & Ok(5) &] \succ \\
[& D(11), & D(11), & Ok(8), & Ok(5) &] \succ \\
[& D(7), & D(7), & Ok(4), & Ok(5) &] \succ \\
[& D(6), & D(6), & Halt(4), & Ok(5) &] \not\succ \\
[& D(12), & D(12), & D(12), & Ok(9) &] \succ \\
[& D(0), & D(0), & D(0), & Exception &]
\end{array}$$

However, with this second semantics, finding a satisfying termination order is easier. The termination ordering is a lexicographic combination of an order comparing the sum of all gas in the frames, an order comparing the size of the call stack, and finally an order comparing the type of the frame (where $Ok > Halt > Exception$). See (EFSimplem, 2020) for the complete formalization and Isabelle/HOL proof. The Isabelle/HOL development is around 900 lines. The proof of soundness is very similar to the previous one. The proof of termination is composed of 14 intermediate lemmas and is around 130 lines.

7 Conclusion

Termination is an important property of any smart contract. To this end, the Ethereum platform (EVM) has introduced a mechanism based on gas which gets consumed as the execution progresses. This paper presents an abstract model of EVM execution that focuses on gas consumption. On this model, we prove that for any EVM execution, gas is used in such a way that it is impossible to construct an infinite loop that does not consume any gas. This property is not immediate to establish for the specification in the EVM Yellow Paper, because of the decidedly nontrivial semantics of contract calls and the fact that cashing-in of the cost of the call is delayed until after the return (whether regular or exceptional).

The proof relies on a non-trivial measure on contract call stacks and has the salient feature that it is independent of the specific costing of instructions, as long as they are greater than 0. This latter point is important as the costs of certain instructions of the EVM has evolved over its rather short life.

The mechanized proof is based on an abstract model of the EVM and fills a gap in current formal developments on verification of contracts with proof assistants (Hirai, 2017; Amani et al., 2018). There are a number of steps for further work related to this mechanization. First, it would be worthwhile formalizing the relation to the complete semantic formalization by Grishchenko (Grishchenko et al., 2018a) or even the Isabelle/HOL formalization (Hirai, 2017; Amani et al., 2018). This can likely be done by setting up a simulation relation between the concrete and the abstract semantics. Second, it would be useful to show that the gas consumption in the two semantics are similar or, at least, that the consumption of one is bounded by a polynomial function of the consumption of the other. Another possible extension stems from the fact that this proof is only for one transaction. It does not take into account several transaction rounds.

Acknowledgements

Many thanks to Emmanuelle Anceaume, Vincent Laporte, Romaric Ludinard, and Clara Schneidewind for valuable discusions about Ethereum and SmartContracts.

REFERENCES

Amani, S., Bégel, M., Bortin, M., and Staples, M. (2018). Towards verifying ethereum smart contract bytecode in isabelle/hol. In *CPP'18*, pages 66–77. ACM.

Buterin, V. (2013). Ethereum: A Next-Generation SmartContract and Decentralized Application Platform. `https://github.com/ethereum/wiki/wiki/White-Paper`.

EFSimplem (2020). Ethereum Formal Semantics for Gas Consumption Analysis - Implementations semantics. `https://github.com/thomas-genet/abstEVM/blob/master/abstEvm2.thy`.

EFSyellow (2020). Ethereum Formal Semantics for Gas Consumption Analysis - Yellow paper semantics. `https://github.com/thomas-genet/abstEVM/blob/master/abstEvm.thy`.

ethertrust (2017). Ethertrust - Trustworthy smart contracts. `https://www.netidee.at/ethertrust`.

ETS (2015). Official ethereum test suite. https://github.com/ethereum/tests.

FEL (2018). Formalization of Ethereum Virtual Machine in Lem. https://github.com/pirapira/eth-isabelle.

Gavin, W. (2014). Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper 151, EIP-150 Revision, http://gavwood.com/Paper.pdf.

Gavin, W. (2019). Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper, BYZANTIUM VERSION e7515a3, https://ethereum.github.io/yellowpaper/paper.pdf.

gevm (2014). Official Go implementation of the Ethereum protocol. https://github.com/ethereum/go-ethereum.

Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., and Smaragdakis, Y. (2018). Madmax: surviving out-of-gas conditions in Ethereum smart contracts. *PACMPL*, 2(OOPSLA):116:1–116:27.

Grishchenko, I., Maffei, M., and Schneidewind, C. (2018a). Foundations and Tools for the Static Analysis of Ethereum Smart Contracts. In *CAV'18*, volume 10981 of *LNCS*, pages 51–78. Springer.

Grishchenko, I., Maffei, M., and Schneidewind, C. (2018b). A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *POST'18*, volume 10804 of *LNCS*, pages 243–269. Springer.

Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B.-M., Park, D., Zhang, Y., Stefanescu, A., and Rosu, G. (2018). KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *CSF'18*, pages 204–217. IEEE Computer Society.

Hirai, Y. (2017). Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *FC'17*, volume 10323 of *LNCS*, pages 520–535. Springer.

Hudson, J. (2016a). Upcoming Ethereum Hard Fork. 2016/10/18, https://blog.ethereum.org/2016/10/18/faq-upcoming-ethereum-hard-fork/.

Hudson, J. (2016b). Upcoming Ethereum Hard Fork. 2016/11/22, https://blog.ethereum.org/2016/11/18/hard-fork-no-4-spurious-dragon/.

pevm (2017). A Python implementation of the Ethereum Virtual Machine. https://github.com/ethereum/py-evm.

script (2014). The script Bitcoin Programming Language. https://en.bitcoin.it/wiki/Script.

solidity (2014). The Solidity Language. https://solidity.readthedocs.io/.

vyper (2017). Pythonic Smart Contract Language for the evm. https://vyper.readthedocs.io/.

Yang, R., Murray, T., Rimba, P., and Parampalli, U. (2019). Empirically Analyzing Ethereum's Gas Mechanism. In *2019 IEEE European Symposium on Security and Privacy Workshops*, pages 310–319. IEEE.